

AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications

Michael Backes^{1,2}, Sebastian Gerling¹, Christian Hammer¹,
Matteo Maffei¹, and Philipp von Styp-Rekowsky¹

¹ Saarland University, CISP

² Max Planck Institute for Software Systems (MPI-SWS)

Abstract Android’s success makes it a prominent target for malicious software. However, the user has very limited control over security-relevant operations. This work presents AppGuard, a powerful and flexible security system that overcomes these deficiencies. It enforces user-defined security policies on untrusted Android applications without requiring any changes to a smartphone’s firmware, root access, or the like. Fine-grained and stateful security policies are expressed in a formal specification language, which also supports secrecy requirements. Our system offers complete mediation of security-relevant methods based on callee-site inline reference monitoring and supports widespread deployment. In the experimental analysis we demonstrate the removal of permissions for overly curious apps as well as how to defend against several recent real-world attacks on Android phones. Our technique exhibits very little space and runtime overhead. The utility of AppGuard has already been demonstrated by more than 1,000,000 downloads.

Keywords: Android, Runtime Enforcement, IRM, Security Policies

1 Introduction

The rapidly increasing number of mobile devices creates a vast potential for misuse. Mobile devices store a plethora of information about our personal lives, and GPS, camera, or microphone offer the ability to track us at all times. The always-online nature of mobile devices makes them a clear target for overly curious or maliciously spying apps and Trojan horses. For instance, social network apps were recently criticized for silently uploading the user’s entire contacts onto external servers [17, 42]. While this behavior became publicly known, users are most often not even aware of what an app actually does with their data. Additionally, fixes for security vulnerabilities in the Android OS often take months until they are integrated into vendor-specific OSs. Between Google’s fix with a public vulnerability description and the vendor’s update, an unpatched system becomes the obvious target for exploits.

Android’s security concept is based on isolation of third-party apps and access control [1]. Access to personal information has to be explicitly granted at install time: When installing an app a list of permissions is displayed, which have to be granted in order to install the app. Users can neither dynamically grant and revoke permissions at runtime, nor add restrictions according to their personal needs. Further, users (and often even developers, cf. [23, 26]) usually do not have enough information to judge whether a permission is indeed required.

Contributions. To overcome the aforementioned limitations of Android’s security system, we present a novel policy-based security framework for Android called AppGuard.

- AppGuard takes an untrusted app and user-defined security policies as input and embeds the security monitor into the untrusted app, thereby delivering a secured self-monitoring app.
- Security policies are formalized in an automata-based language that can be configured in AppGuard. Security policies may specify restrictions on method invocations as well as secrecy requirements.
- AppGuard is built upon a novel approach for callee-site inline reference monitoring (IRM). We redirect method calls to the embedded security monitor and check whether executing the call is allowed by the security policy. Technically, this is achieved by altering method references in the Dalvik VM. This approach does not require root access or changes to the underlying Android architecture and, therefore, supports widespread deployment as a stand-alone app. It can handle even JAVA reflection (cf. section 3) and dynamically loaded code.
- Secrecy requirements are enforced by storing the secret within the security monitor. Apps are just provided with a handle to that secret. This mechanism is general enough to enforce the confidentiality of data persistently stored on the device (e.g., address book entries or geolocation) as well as of dynamically received data (e.g., user-provided passwords or session tokens received in a single sign-on protocol). The monitor itself is protected against manipulation of its internal state and forceful extraction of stored secrets.
- We support fully-automatic on-the-phone instrumentation (no root required) of third-party apps and automatic updates of rewritten apps such that no app data is lost. Our system has been downloaded by about 1,000,000 users (aggregated from [5, 12, 34]) and has been invited to the Samsung Apps market.
- Our evaluation on typical Android apps has shown very little overhead in terms of space and runtime. The case studies demonstrate

the effectiveness of our approach: we successfully revoked permissions of excessively curious apps, demonstrate complex policies, and prevent several recent real-world attacks on Android phones, both due to in-app and OS vulnerabilities. We finally show that for the vast majority of 25,000 real-world apps, our instrumentation does not break functionality, thus demonstrating the robustness of our approach.

Key Design Decisions & Closely Related Work. Researchers have proposed several approaches to overcome the limitations of Android’s security system, most of which require modifications to the Android platform. While there is hope that Google will eventually introduce a more fine-grained security system, we decided to directly integrate the security monitor within the apps, thereby requiring no change to the Android platform. The major drawback of modifying the firmware and platform code is that it requires rooting the device, which may void the user’s warranty and affect the system stability. Besides, there is no general Android system but a plethora of vendor-specific variants that would need to be supported and maintained across OS updates. Finally, laymen users typically lack the expertise to conduct firmware modifications, and, therefore, abstain from installing modified Android versions.

Aurasium [45], a recently proposed tool for enforcing security policies in Android apps, rewrites low-level function pointers of the libc library in order to intercept interactions between the app and the OS. A lot of the functionality that is protected by Android’s permission system depends on such system calls and thus can be intercepted at this level. A limitation of this approach is that the parameters of the original Java requests need to be recovered from the system calls’ low-level byte arrays in order to differentiate malicious requests from benign ones, which “is generally difficult to write and test” [45] and may break in the next version of Android at Google’s discretion. Similarly, mock return values are difficult to inject at this low level. In contrast, we designed our system to intercept high-level Java calls, which allows for more flexible policies. In particular we are able to inject arbitrary mock return values, e.g. a proxy object that only gives access to certain data, in case of policy violations. Additionally, we are able to intercept security-relevant methods that do not depend on the libc library. As an example consider the policy that systematically replaces MD5, which is nowadays widely considered an insecure hashing algorithm, by SHA-1. Since the implementation of MD5 does not use any security-relevant functionality of the libc library, this policy cannot be expressed in Aurasium. Finally, it is worth to mention that both Aurasium and AppGuard offer only limited guarantees for

Table 1. Comparison of Android IRM approaches

Feature	1	2	3	4	5	6	7	8	Runtime Overhead
Aurasium [45]	✓	–	I	●	✓	–	–	–	14-35%
Dr. Android [35]	✓	–	E	●	●	–	–	–	10-50%
I-ARM-Droid [15]	✓	–	I	–	●	–	–	✓	16%
AppGuard	✓	✓	I	●	✓	✓	✓	✓	1-21%

Legend: 1. No Firmware Mod. 2. On Phone Instr./Updates 3. Monitor
 4. Native Methods 5. Reflection 6. Policy Lang. 7. Data Secrecy
 8. Parametric Joinpoints; ✓: full support, ●: partial support

apps incorporating native code. Aurasium can detect an app that tries to perform security-relevant operations directly from native code, under the assumption, however, that the code does not re-implement the libc functionality. Our approach can monitor Java methods invoked from native code, although it cannot monitor system calls from native code.

Jeon et al. [35] advocate to place the reference monitor into a separate application. Their approach removes all permissions from the monitored app, as all calls to sensitive functionality are done in the monitoring app. This is fail-safe by default as it prevents both reflection and native code from executing such functionality. However, it has some drawbacks: If a security policy depends on the state of the monitored app, this approach incurs high complexity and overhead as all relevant data must be marshaled to the monitor. Besides, the monitor may not yet be initialized when the app attempts to perform security-relevant operations. Finally, this approach does not follow the principle of least privilege since the monitor must have the permissions of all monitored apps, making it a prominent target for privilege escalation attacks [9]. We propose a different approach: Although the security policies are specified and stored within AppGuard, the policy enforcement mechanism is directly integrated and performed within the monitored apps. The policy configuration file is passed as input to the security monitor embedded in each app, thereby enabling dynamic policy configuration updates. This approach does not involve any inter-procedure calls and obeys the principle of least privilege, as AppGuard requires no special permissions. Hence, AppGuard is not prone to privilege escalation attacks.

Table 1 compares AppGuard with the most relevant related work that does not modify the firmware. Up to now, no other system can instrument an app and update apps directly on the phone. Dr. Android has an external monitor (E) accessed via IPC; the other three approaches

use internal monitors (I). Aurasium can monitor security-relevant native methods, Dr. Android only removes their permissions, which may lead to unexpected program termination, whereas our tool can prevent calls to sensitive Java APIs from native code. Both Aurasium and AppGuard handle reflection; Dr. Android does not handle it; I-ARM-Droid prevents it altogether. AppGuard is the only system that offers a high-level specification language for policies and supports hiding of secret data from e.g. untrusted components in the monitored app. Both Aurasium and Dr. Android only support a fixed set of joinpoints where a security policy can be attached to. In contrast, I-ARM-Droid and AppGuard can instrument calls to any Java method. The last column displays the runtime overhead incurred in micro-benchmarks as reported by the respective authors. AppGuard is competitive in terms of runtime overhead with respect to concurrent efforts. In our previous work [44] we presented the initial idea for diverting method calls in the Dalvik VM with a rudimentary implementation for micro-benchmarks only. It did *not* support a policy language, secrecy, and on-the-phone instrumentation, and did not include case studies. A recent tool paper [3] presented a previous version of AppGuard, which is based on caller-site instrumentation.

2 AppGuard

Runtime policy enforcement for third-party apps is challenging on unmodified Android systems. Android’s security concept strictly isolates different apps installed on the same device. Communication between apps is only possible via Android’s inter-process communication (IPC) mechanism. However, such communication requires both parties to cooperate, rendering this channel unsuitable for a generic runtime monitor. Apps cannot gain elevated privileges to observe the behavior of other apps.

AppGuard tackles this problem by following an approach pioneered by Erlingsson and Schneider [21] called *inline reference monitor* (IRM). The basic idea is to rewrite an untrusted app such that the code that monitors the app is directly embedded into its code. To this end, IRM systems incorporate a *rewriter* or *inliner* component, that injects additional security checks at critical points into the app’s bytecode. This enables the monitor to observe a trace of *security-relevant events*, which typically correspond to invocations of trusted system library methods from the untrusted app. To actually enforce a *security policy*, the monitor controls the execution of the app by suppressing or altering calls to security-relevant methods, or even terminating the program if necessary.

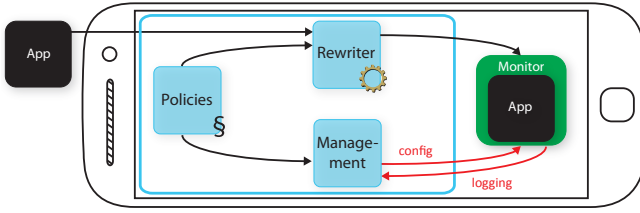


Figure 1. Schematics of AppGuard

In the IRM context, a policy is typically specified by means of a security automaton that defines which sequences of security-relevant events are acceptable. Such policies have been shown to express exactly the policies enforceable by runtime monitoring [43]. Ligatti et al. differentiate security automata by their ability to enforce policies by manipulating the trace of the program [37]. Some IRM systems [16, 21] implement truncation automata, which can only terminate the program if it deviates from the policy. However, this is often undesirable in practice. *Edit automata* [37] transform the program trace by inserting or suppressing events. Monitors based on edit automata are able to react gracefully to policy violations, e.g., by suppressing an undesired method call and returning a mock value, thus allowing the program to continue.

AppGuard is an IRM system for Android with the transformation capabilities of an edit automaton. Figure 1 provides a high-level overview of our system. We distinguish three main components:

1. *A set of security policies.* On top of user-defined and app-specific policies, AppGuard provides various generic security policies that govern access to platform API methods which are protected by coarse-grained Android permissions. These methods comprise, e.g., methods for reading personal data, creating network sockets, or accessing device hardware like the GPS or the camera. As a starting point for the security policies, we used a mapping from API methods to permissions [23].
2. *The program rewriter.* Android apps run within a register-based Java VM called *Dalvik*. Our rewriter manipulates Dalvik executable (*dex*) bytecode of untrusted Android apps and embeds the security monitor into the untrusted app. The references of the Dalvik VM are altered so as to redirect the method calls to the security monitor.
3. *A management component.* AppGuard offers a graphical user interface that allows the user to set individual policy configurations on a per-app basis. In particular, policies can be turned on or off and parameterized. In addition, the management component keeps a de-

tailed log of all security-relevant events, enabling the user to monitor the behavior of an app.

3 Architecture

AppGuard [5] is a stand-alone Android app written in Java and C that comprises about 9000 lines of code. It builds upon the *dexlib* library, which is part of the *smali* disassembler for Android by Ben Gruver [30], for manipulating *dex* files. The size of the app package is roughly 2 Mb.

Instrumentation. IRM systems instrument a target app such that the control flow of the program is diverted to the security monitor whenever a security-relevant method is about to be invoked. There are two strategies for passing control to the monitor: Either at the call-site in the app code, right before the invocation of the security-relevant method, or at the callee-site, i.e. at the beginning of the security-relevant method. The latter strategy is simpler and more efficient, because callee sites are easily identified and less in number [7]. Furthermore, callee-site rewriting can handle obfuscated apps as it does not require to “understand” the untrusted code. Unfortunately, in our setting, standard callee-site rewriting is not feasible for almost all security-relevant methods, as they are defined in Android system libraries, which cannot be modified.

In order to achieve the same effect as callee-site rewriting, AppGuard uses a novel dynamic call-interposition approach [44]. It diverts calls to security-relevant methods to functions in the monitor (called *guards*) that perform a security check. In order to divert the control flow we replace the reference to a method’s bytecode in the VM’s internal representation (e.g., a virtual method table) with the reference to our security guard. The security guards reside in an external library that is dynamically loaded on app startup. Therefore, we do not need to reinstrument the app when a security policy is modified. Additionally, we store the original reference in order to access the original function later on, e.g., in case the security check grants the permission to execute the security-critical method. This procedure also reduces the risk of accidentally introducing infinite loops by a policy since we usually call the original method.

With this approach, invocations of security-relevant methods do *not* need to be rewritten statically. Instead, we use Java Native Interface (JNI) calls at runtime to replace the references to each of the monitored functions. More precisely, we call the JNI method `GetMethodID()` which takes a method’s signature, and returns a pointer to the internal data structure describing that method. This data structure contains a refer-

```

public class Main {
    public static void main(String[] args) {
        A.foo(); // calls A.foo()
        MethodHandle A_foo = Instrumentation.replaceMethod(
            "Lcom/test/A;->foo()", "Lcom/test/B;->bar()");
        A.foo(); // calls B.bar()
        Instrumentation.callOriginalMethod(A_foo); // calls A.foo()
    }
}

```

Figure 2. Example illustrating the functionality of the instrumentation library

ence to the bytecode instructions associated with the method, as well as metadata such as the method’s argument types or the number of registers. In order to redirect the control flow to our guard method, we overwrite the reference to the instructions such that it points to the instructions of the security guard’s method instead. Additionally, we adjust the intercepted method’s metadata (e.g., number of registers) to be compatible with the guard method’s code. This approach works both for pure Java methods and methods with a native implementation.

Figure 2 illustrates how to redirect a method call using our instrumentation library. Calling `Instrumentation.replaceMethod()` replaces the instruction reference of method `foo()` of class `com.test.A` with the reference to the instructions of method `bar()` of class `com.test.B`. It returns the original reference, which we store in a variable `A_foo`. Calling `A.foo()` will now invoke `B.bar()` instead. The original method can still be invoked by `Instrumentation.callOriginalMethod(A_foo)`. Note that the handle `A_foo` will be a secret of the security monitor in practice. Therefore the original method can no longer be invoked directly by the instrumented app.

Policies. We developed a high-level policy language called SOSPoX in order to express and characterize the security policies supported by AppGuard. SOSPoX is based on SPoX [31, 32] and is a direct encoding of edit automata. SOSPoX policies enable the specification of constraints on the execution of method calls as well as changes of the control flow. This includes the specification of a graceful reaction to policy violations, e.g., by suppressing an undesired method call and returning a mock value, thus allowing the program to continue. Furthermore, SOSPoX offers support for confidentiality policies. Data returned by method invocations are labeled as either confidential or public: confidential data can only be processed by the methods authorized by the policy. In general, we can specify information flow policies that prevent both explicit flows (i.e.,

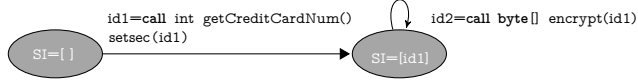


Figure 3. Security automaton exemplifying declassification by encryption

through assignments) and implicit flows (i.e., through the control flow of the program). This can be achieved by a policy disallowing the processing of confidential data. Declassification policies allow selected methods to process confidential data and the returned results are labeled as public. For instance, we can specify that the return value of a function that returns our credit card number is to be kept secret, but that the encryption of the returned credit card number counts as declassification and is no longer secret (cf. Figure 3). Due to space constraints we omit the technical details of our policies and refer to [2] for a more comprehensive presentation and additional policy examples.

Rewriter. The task of the rewriter component is to insert code into the target app, which dynamically loads the monitor package into the app’s virtual machine. To ensure instrumentation of security-sensitive methods before their execution, we create an application class that becomes the superclass of the existing application class³. Our new class contains a static initializer, which becomes the very first code executed upon app startup. The initializer uses a custom class loader to load our monitor package. Afterwards, it calls an initializer method in the monitor that uses the instrumentation library to rewrite the method references.

Separation of Secrets. Policies in our system can specify that the return values of certain functions are to be kept secret. In order to prevent an app from leaking secret values, we control access to these secrets. To this end, the monitor intercepts all calls to methods that the policy annotates as “secret-carrying”, i.e. methods that can produce secret output or receive secret input. Whenever the invocation of such a method produces a new secret output, the monitor returns a dummy value, which serves as a reference to the secret for further processing. If such a secret reference is passed to a method that supports secret parameters, the trampoline method invokes the original method with the corresponding secret instead and returns either the actual result or a new secret reference, in case the return value was marked as secret in the policy. The dummy reference values do not contain any information about the secret itself and are thus innocuous if processed by any method that is not annotated in the policy.

³ In case no application class exists, we register our class as the application class.

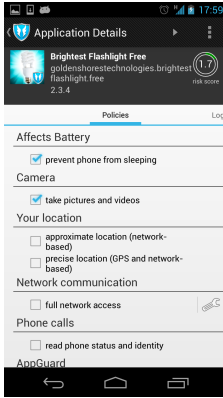


Figure 4. Permission configuration for the Tiny Flashlight app

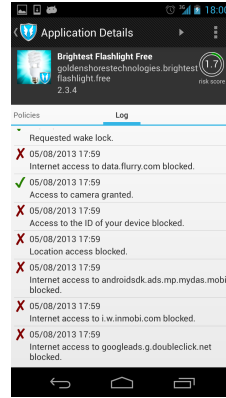


Figure 5. Log of security-relevant operations

Management. The management component of AppGuard monitors the behavior of instrumented apps and offers policy configuration at runtime. This configuration is provided to the instrumented app as a world-readable file. Its location is hardcoded into the monitor during the rewriting process. This is motivated by the fact that invocations of security-relevant methods can occur before the management app is fully initialized and able to react on Android IPC. The management component provides a log of all security-relevant method invocations for each app, which enables the user to make informed decisions about the current policy configuration. Invocations are reported to the management app using a standard Android Service component. The asynchronous nature of Android IPC is not an issue, since security-relevant method invocations that occur before the service connection is established are buffered locally.

Monitor Protection. In our system, the inlined monitor is part of the monitored app. A malicious app might try to circumvent the monitor by tampering with its internal state. Furthermore, an app could try to subvert secrecy policies by directly extracting stored secrets from the monitor. Since the monitor package containing secret data and pointers to the original methods is unknown at compile time and due to strong typing, a malicious app would need to rely on reflection to access the monitor. To thwart such attacks, we implement a `ReflectionPolicy` that intercepts function calls to the `Reflection` API. In particular, we monitor operations that access Java classes and fields like `java.lang.Class->forName()` or `java.lang.Class->getField()` and thereby effectively prevent access to the monitor package.

Deployment. On unmodified Android systems, app sandboxing prevents direct modifications of the code of other apps installed on the device. AppGuard leverages the fact that the app packages of installed third-party apps are stored in a world-readable location in the filesystem. Thus the monitor is capable of inlining any app installed on the device by processing the corresponding apk file. In the end, AppGuard produces a self-monitoring app package that replaces the original version. Since stock Android does not allow automatic (un)installation of other apps, the user is prompted to confirm both the removal of the original app as well as the installation of the instrumented app. Moreover, we ask the user to enable the OS-option “Unknown sources: Allow installation of apps from sources other than the Play Store”. Due to these two user interactions, no root privileges are required for AppGuard.

All Android apps need to be signed with a developer key. Since our rewriting process breaks the original signature, we sign the modified app with a new key. Apps signed with the same key can access each other’s data if they declare so in their manifests. Thus, we sign rewritten apps with keys based on their original signatures in order to preserve the original behavior. In particular, two apps that were originally signed with the same key, are signed with the same new key after the rewriting process.

Finally, due to the different signature, instrumented apps would no longer receive automatic updates, which may negatively impact device security. Therefore, AppGuard assumes the role of the Play Store app and checks for updates of instrumented apps. If a new version is found, AppGuard prompts to download the app package, instruments it and replaces the existing version of the app.

4 Experimental Evaluation

In this section we present the results of our experimental evaluation. We used a Google Galaxy Nexus smartphone (1.2 GHz, two cores, 1GB RAM) with Android 4.1.2 for on-the-phone evaluations and a notebook with an Intel Core i5-2520M CPU (2.5 GHz, two cores, hyper-threading) and 8GB RAM for off-the-phone evaluations.

4.1 Robustness and Performance Evaluation

Robustness. We tested AppGuard on more than 25,000 apps from two different app markets and report the results in Table 2. The stability of the original apps is tested using the UI/Application Exerciser Monkey

Table 2. Robustness of rewriting and monitoring

App Market	Apps	Stable	Dex verified	Stable Instr.
Google Play	9508	8783	9508 (100%)	8744 (99.6%)
SlideMe	15974	14590	15974 (100%)	14469 (99.1%)
Total	25482	23373	25482 (100%)	23213 (99.3%)

provided by the Android framework with a random seed and 1000 injected events (third column). To evaluate the robustness of the rewriting process we check the validity of the generated `dex` file (fourth column) and test the stability of the instrumented app using the UI Monkey with the random seed (fifth column). Note that we only consider the stability of instrumented apps where the original did not crash.

The reported numbers indicate a very high reliability of the instrumentation process: we found no illegal `dex` file and over 99% of the stable apps were also stable after the instrumentation. The majority of the remaining 1% does not handle checked exceptions gracefully (e.g. `IOException`), which may be thrown by AppGuard when suppressing a function call. This bad coding style is not found in popular apps. Other apps terminate when they detect a different app signature. In rare cases, the mock values returned by suppressed function calls violate an invariant of the program. Note, however, that our test with the UI Monkey does not check for semantic equivalence.

Performance. AppGuard modifies apps installed on an Android device by adding code at the bytecode level. We analyze the time it takes to rewrite an app and its impact on both size and execution time of the modified app. Table 3 provides an overview of our performance evaluation for the rewriting process. We tested AppGuard with 8 apps and list the following results for each of the apps: size of the original app package (Apk), size of the `classes.dex` file, and the duration of the rewriting process both on the laptop and smartphone (PC and Phone, respectively).

The size of the `classes.dex` file increases on average by approximately 3.7 Kb. This increase results from merging code that loads the monitor package into the app. Since we perform callee-site rewriting and load the our external policies dynamically, we only have this static and no proportional increase of the original dex file. For a few apps (e.g. Angry Birds) the instrumentation time is dominated by re-building and compressing the app package file (which is essentially a zip archive). The evaluation also clearly reveals the difference in computing power between the laptop

Table 3. Sizes of apk and dex files with rewriting time on PC and phone.

App (Version)	Size [Kb]		Time [sec]	
	Apk	Dex	PC	Phone
Angry Birds (2.0.2)	15018	994	5.8	39.3
Endomondo (7.0.2)	3263	1635	0.7	16.6
Facebook (1.8.3)	4013	2695	1.2	26.4
Instagram (1.0.3)	12901	3292	3.0	44.3
Tiny Flashlight (4.7)	1287	485	0.1	2.9
Twitter (3.0.1)	2218	764	0.3	8.9
Wetter.com (1.3.1)	4296	958	0.4	10.7
WhatsApp (2.7.3581)	5155	3182	0.8	27.7

Table 4. Runtime comparison with micro-benchmarks for normal function calls and guarded function calls with policies disabled as well as the introduced runtime overhead.

Function Call	Original Call	Guarded Call	Overhead
Socket-><init>()	0.0186 ms	0.0212 ms	21.4%
ContentResolver->query()	19.5229 ms	19.4987 ms	0.8%
Camera->open()	74.498 ms	79.476 ms	6.4%

and the phone. While the rewriting process takes considerably more time on the phone than on the laptop, we argue that this should not be a major concern as the rewriter is only run once per app.

The runtime overhead introduced by the inline reference monitor is measured through micro-benchmarks (cf. Table 4.) We compare the execution time of single function calls in three different settings: the original code with no instrumentation, the instrumented code with disabled policies (i.e. policy enforcement turned off.), and the incurred overhead. We list the average execution time for each function call. For all function calls the instrumentation adds a small runtime overhead due to additional code. If we enabled policies, the changed control flow usually leads to shorter execution times and renders them incomparable. Even with disabled policies the incurred runtime overhead is negligible and does not adversely affect the app’s performance.

4.2 Case Study Evaluation

We evaluate our framework in several case studies by applying different policies to real world apps from Google Play [28] (cf. Table 3 for the analyzed versions). As a disclaimer, we would like to point out that we use

apps from the market for exemplary purposes only, without implications regarding their security unless we state this explicitly.

For our evaluation, we implemented 9 different policies. Five of them are designed to revoke critical Android platform permissions, in particular the Internet permission (`InternetPolicy`), access to camera and audio hardware (`CameraPolicy`, `AudioPolicy`), and permissions to read contacts and calendar entries (`ContactsPolicy`, `CalendarPolicy`). Furthermore, we introduce a complex policy that tracks possible fees incurred by untrusted applications (`CostPolicy`). The `HttpsRedirectPolicy` and `MediaStorePolicy` address security issues in third-party apps and the OS. Finally, the `ReflectionPolicy` described in section 3 monitors invocations of Java’s Reflection API and an app-specific policy. In the following case studies, we highlight 7 of these policies and evaluate them in detail on real-world apps.

Our case studies focus on (a) the possibility to revoke standard Android permissions. Additionally, it is possible to (b) enforce fine-grained policies that are not supported by Android’s existing permission system. Our framework provides quick-fixes and mitigation for vulnerabilities both in (c) third-party apps and (d) the operating system⁴. Finally, we present a general security policy that is completely independent of Android’s permission system.

Revoking Android permissions. Many Android applications request more permissions than necessary. AppGuard gives users the chance to safely revoke permissions at any time at a fine-grained level.

Case study: Twitter. As an example for the revocation of permissions, we chose the official app of the popular micro-blogging service Twitter. It attracted attention in the media [42] for secretly uploading phone numbers and email addresses stored in the user’s address book to the Twitter servers. While the app “officially” requests the permissions to access both Internet and the user’s contact data, it did not indicate that this data would be copied off the phone as part of the “Find friends” feature that makes friend suggestions based on the user’s address book. As a result of the public disclosure, the current version of the app now explicitly informs the user before uploading any personal information.

To prevent leakage of private information, we block access to the user’s contact list. Since friends can also be added manually, AppGuard’s `ContactsPolicy` protects the user’s privacy while losing only minor convenience functionality. The actual policy enforcement is done by monitoring queries to the `ContentResolver`, which serves as a centralized access point

⁴ By providing policy recommendations based on a crowdsourcing approach, even laymen users can enforce complex policies (e.g. to fix OS vulnerabilities)

to Android’s various databases. Data is identified by a URI, which we examine to selectively block queries to the contact list by returning a mock result object. Our tests were carried out on an older version of the Twitter app, which was released prior to their fix.

Case study: Tiny Flashlight. The app either uses the camera’s flash LED as a flashlight, or turns the whole screen white and requests the permissions to access the Internet and the camera. Manual analysis indicates that the Internet permission is only required to display online ads. However, together with the camera, this app could potentially be abused for spying purposes, which would be hard to detect without detailed code or traffic analysis. AppGuard can block Internet access of the app with the `InternetPolicy`, which blocks the in-app ads. We monitor constructor calls of the various `Socket` classes, the `java.net.url.openConnection()` method as well as several other network I/O functions, and throw an `IOException` if access to the Internet is forbidden.

Enforcing fine-grained policies. AppGuard can also add new restrictions to functionalities that are not restricted by the current permission system or that are already protected, but not in the desired way. For example, from the user’s point of view most apps should only communicate with a limited set of servers. The `wetter.com` app provides weather information and should only communicate with its servers to query weather information. The `InternetPolicy` of AppGuard provides fine grained Internet access based on per-app white-listing of web servers. For this app we restrict Internet access with the regular-expression `^(.+\.)?wetter\.com$`, which blocks potentially harmful connections to other servers. White-listing can be configured in the management interface by selecting from a list of hosts the app has already attempted to connect to.

Quick-fixes for vulnerabilities in third-party apps. Although most apps use encrypted `https` for the login procedures to web servers, there are apps that return to unencrypted `http` after successful login, thereby transmitting their authentication tokens in plain text over the Internet. Attackers could eavesdrop on the connection to impersonate the user [36].

Endomondo Sports Tracker returns to `http` after successful login, thereby leaking the authentication token. As the Web server supports `https` for the whole session, the `HttpsRedirectPolicy` of AppGuard enforces the permanent usage of `https`, which protects the user’s account and data from identity theft. Depending on the monitored function, we return the redirected `https` connection or the content from the redirected connection.

Mitigation for operating system vulnerabilities. We also found our tool useful to mitigate operating system vulnerabilities. As we cannot

change the operating system itself, we instrument all apps with a global security policy to prevent exploits. For example, Android apps do not require a special permission to access the photo storage. Any app with the Internet permission could thus leak private photos without the user’s knowledge. We address this problem with a global `MediaStorePolicy` policy that monitors calls to the `ContentResolver` object. Moreover, any app could use the Android browser to leak arbitrary data, by sending an appropriate Intent. The `InternetPolicy` monitors the `startActivity(Intent)` calls and throws an exception if the particular intent is not allowed. It thereby also prevents the local cross-site scripting attack [4] against the Android browser that was present up to Android 2.3.4. Using a combination of `VIEW` intents, it was possible to trick the browser into executing arbitrary JavaScript code within the domain of the attacker’s choice, which enabled the attacker to steal login information or even silently install additional apps.

Threats to Validity. Like any IRM system, AppGuard’s monitor runs within the same process as the target app. This makes it vulnerable to attacks from malicious apps that try to bypass or disable the security monitor. Our instrumentation technique is robust against attacks from Java code, as this code is strongly typed. It can handle cases like reflection or dynamically loaded libraries. However, a malicious app could use native code to disable the security monitor by altering the references we modified or tampering with the AppGuard’s bytecode instructions or data structures. To prevent this, we could block the execution of any untrusted native code by intercepting calls to `System.loadLibrary()`, which is, however, not a viable solution in practice. Currently, AppGuard warns the user if an app attempts to execute untrusted native code.

In order to assess the potential impact of native code on our approach, we analyzed the percentage of apps that rely on it. Our evaluation [2] on 25,000 apps (cf. Table 5) revealed that about 15% include native libraries, which is high compared to the 5% of apps reported in [46]. We conjecture that this difference is due to the composition of our sample. It consists of 30% games, which on Android frequently build upon native code based game engines (e.g., `libGDX` or `Unity`) to improve performance. Ignoring games, we found only 9% of the apps to be using native code, which makes AppGuard a safe solution for over 90% of these apps.

AppGuard monitors the invocation of security-relevant methods, which are typically part of the Android framework API. By reimplementing parts of this API and directly calling into lower layers of the framework, a malicious app could circumvent the security monitor. This attack vector is always available to attackers in IRM systems that monitor method

Table 5. Ratio of apps using native code

App Market	Overall		Games		No games	
	Apps	Nat. code	Apps	Nat. code	Apps	Nat. code
Google Play	9508	2212 (23%)	2838	1110 (39%)	6670	1102 (16%)
SlideMe	15974	1693 (10%)	5920	1244 (21%)	10054	449 (4.5%)
Total	25482	3905 (15%)	8758	2354 (26%)	16724	1551 (9.2%)

invocations. Furthermore, AppGuard is not designed to be stealthy: due to the resigning of apps, instrumentation transparency cannot be guaranteed. There are many apps that verify their own signature (e.g. from the Amazon AppStore). If they rely on Android API to retrieve their own signature, however, AppGuard can hook these functions to return the original signature, thus concealing its presence. An app could also detect the presence of AppGuard by looking for the presence of AppGuard classes in the virtual machine. In the end, both of these attacks boil down to an arms race, that a determined attacker will win. Up to now, we did not detect any app that tried to explicitly circumvent AppGuard.

Our instrumentation approach relies only on the layout of Dalvik’s internal data structure for methods, which has not changed since the initial version of Android. However, our instrumentation system could easily be adapted if the layout were to change in future versions of Android.

Android programs are multi-threaded by default. Issues of thread safety could therefore arise in the monitor when considering stateful policies that take the relative timing of events in different threads into account. While we did not yet experiment with such policies, we plan to extend our system to support *race-free policies* [14] in the future. In contrast, policies that atomically decide whether to permit a method call are also correct in the multithreaded setting.

5 Further Related Work

Researchers have worked on various security aspects of Android and proposed many security enhancements. One line of research [10, 18, 19, 27, 41] targets the detection of privacy leaks and malicious third-party apps. Another line of work analyzed Android’s permission based access control system. Barrera et al. [6] conducted an empirical analysis of Android’s permission system on 1,100 Android apps and suggested improvements to its granularity. Felt et al. [24] analyzed the effectiveness of app permissions using case studies on Google Chrome extensions and Android apps.

The inflexible and coarse-grained permission system of Android inspired many researchers to propose extensions [20, 29, 38–40]. Conti et al. [13] integrate a context-related policy enforcement mechanism into Android. Fragkaki et al. [25] present an external reference monitor approach to enforce coarse grained secrecy and integrity policies. In contrast, our intention was to deploy the system to unmodified stock Android phones.

The concept of IRMs has received considerable attention in the literature. It was first formalized by Erlingsson and Schneider in the development of the SASI/PoET/PSLang systems [21, 22], which implement IRM’s for x86 assembly code and Java bytecode. Several other IRM implementations for Java followed. Polymer [8] is a IRM system based on edit automata, which supports composition of complex security policies from simple building blocks. The Java-MOP [11] system offers a rich set of formal policy specification languages. IRM systems have also been developed for other platforms. Mobile [33] is an extension to Microsoft’s .NET Common Intermediate Language that supports certified IRM. Finally, the S3MS.NET Run Time Monitor [16] enforces security policies expressed in a variety of policy languages for .NET applications.

6 Conclusions

We presented a practical approach to enforce high-level, fine-grained security policies on stock android phones. It is built upon a novel approach for callee-site inline reference monitoring and provides a powerful framework for enforcing arbitrary security and secrecy policies. Our system instruments directly on the phone and allows automatic updates without losing user data. The system curbs the pervasive overly curious behavior of Android apps. We enforce complex stateful security policies and mitigate vulnerabilities of both third-party apps and the OS. AppGuard goes even one step beyond being capable of efficiently protecting secret data from misuse in untrusted apps. Our experimental analysis demonstrates the robustness of the approach and shows that the overhead in terms of space and runtime are negligible. The case studies illustrate how AppGuard prevents several real-world attacks on Android. A recent release of AppGuard has already been downloaded by more than 1,000,000 users.

Acknowledgement. We thank the anonymous reviewers for their comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and both the initiative for excellence and the Emmy Noether program of the German federal government.

References

1. Android.com: Security and Permissions (2012), <http://developer.android.com/guide/topics/security/security.html>
2. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard – Fine-grained Policy Enforcement for Untrusted Android Applications. Tech. Rep. A/02/2013, Saarland University (April 2013)
3. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard - Enforcing User Requirements on Android Apps. In: Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013) (2013), to appear.
4. Backes, M., Gerling, S., von Styp-Rekowsky, P.: A Local Cross-Site Scripting Attack against Android Phones (2011), http://www.infsec.cs.uni-saarland.de/projects/android-vuln/android_xss.pdf
5. Backes SRT: SRT AppGuard : Mobile Android Security Solution, <http://www.srt-appguard.com/en/>
6. Barrera, D., Kayacık, H.G., van Oorschot, P.C., Somayaji, A.: A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In: Proc. 17th ACM Conference on Computer and Communication Security (CCS 2010). pp. 73–84 (2010)
7. Bauer, L., Ligatti, J., Walker, D.: A Language and System for Composing Security Policies. Tech. Rep. TR-699-04, Princeton University (January 2004)
8. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005). pp. 305–314 (2005)
9. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastri, B.: Towards Taming Privilege-Escalation Attacks on Android. In: Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2012) (2012)
10. Chaudhuri, A., Fuchs, A., Foster, J.: SCanDroid: Automated Security Certification of Android Applications. Tech. Rep. CS-TR-4991, University of Maryland (2009), <http://www.cs.umd.edu/~avik/papers/scandroidasca.pdf>
11. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Proc. 11th International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS 2005). vol. 3440, pp. 546–550. Springer-Verlag (2005)
12. Chip: SRT AppGuard, http://www.chip.de/downloads/SRT-AppGuard-Android-App_56552141.html
13. Conti, M., Nguyen, V.T.N., Crispo, B.: CRePE: Context-Related Policy Enforcement for Android. In: Proc. 13th International Conference on Information Security (ISC 2010). pp. 331–345 (2010)
14. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Security Monitor Inlining and Certification for Multithreaded Java. Mathematical Structures in Computer Science (2011)
15. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In: Mobile Security Technologies 2012 (MoST 12) (2012)
16. Desmet, L., Joosen, W., Massacci, F., Naliuka, K., Philippaerts, P., Piessens, F., Vanoverberghe, D.: The S3MS.NET Run Time Monitor. Electron. Notes Theor. Comput. Sci. 253(5), 153–159 (Dec 2009)

17. von Eitzen, C.: Apple: Future iOS release will require user permission for apps to access address book (February 2012), <http://h-online.com/-1435404>
18. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proc. 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI 2010). pp. 393–407 (2010)
19. Enck, W., Ocateau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: Proc. 20th Usenix Security Symposium (2011)
20. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proc. 16th ACM Conference on Computer and Communication Security (CCS 2009). pp. 235–245 (2009)
21. Erlingsson, Ú., Schneider, F.B.: IRM Enforcement of Java Stack Inspection. In: Proc. 2002 IEEE Symposium on Security and Privacy (Oakland 2002). pp. 246–255 (2000)
22. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proc. of the 1999 workshop on New security paradigms (NSPW 1999). pp. 87–95 (2000)
23. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android Permissions Demystified. In: Proc. 18th ACM Conference on Computer and Communication Security (CCS 2011) (2011)
24. Felt, A.P., Greenwood, K., Wagner, D.: The Effectiveness of Application Permissions. In: Proc. 2nd Usenix Conference on Web Application Development (WebApps 2011) (2011)
25. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and Enhancing Android’s Permission System. In: Proc. 17th European Symposium on Research in Computer Security (ESORICS 2012) (2012)
26. Gibler, C., Crussel, J., Erickson, J., Chen, H.: AndroidLeaks: Detecting Privacy Leaks in Android Applications. Tech. Rep. CSE-2011-10, University of California Davis (2011)
27. Gilbert, P., Chun, B.G., Cox, L.P., Jung, J.: Vision: Automated Security Validation of Mobile Apps at App Markets. In: Proc. 2nd International Workshop on Mobile Cloud Computing and Services (MCS 2011) (2011)
28. Google Play (2012), <https://play.google.com/store>
29. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic Detection of Capability Leaks in Stock Android Smartphones. In: Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2012) (2012)
30. Gruver, B.: Smali: A assembler/disassembler for Android’s dex format, <http://code.google.com/p/smali/>
31. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: Proc. 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2008). pp. 11–20 (2008)
32. Hamlen, K.W., Jones, M.M., Sridhar, M.: Chekov: Aspect-oriented Runtime Monitor Certification via Model-checking. Tech. Rep. UTDCS-16-11, University of Texas at Dallas (May 2011)
33. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified In-lined Reference Monitoring on .NET. In: Proc. 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2006). pp. 7–16 (2006)
34. Heise: SRT AppGuard, <http://www.heise.de/download/srt-appguard-pro-1187469.html>

35. Jeon, J., Micinski, K.K., Vaughan, J.A., Reddy, N., Zhu, Y., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. Tech. Rep. CS-TR-5006, University of Maryland (December 2011)
36. Könings, B., Nickels, J., Schaub, F.: Catching AuthTokens in the Wild - The Insecurity of Google's ClientLogin Protocol. Tech. rep., Ulm University (2011), <http://www.uni-ulm.de/in/mi/mi-mitarbeiter/koenings/catching-authtokens.html>
37. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security* 4(1–2), 2–16 (2005)
38. Nauman, M., Khan, S., Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2010)*. pp. 328–332 (2010)
39. Ongtang, M., Butler, K.R.B., McDaniel, P.D.: Porscha: policy oriented secure content handling in Android. In: *Proc. 26th Annual Computer Security Applications Conference (ACSAC 2010)*. pp. 221–230 (2010)
40. Ongtang, M., McLaughlin, S.E., Enck, W., McDaniel, P.: Semantically Rich Application-Centric Security in Android. In: *Proc. 25th Annual Computer Security Applications Conference (ACSAC 2009)*. pp. 340–349 (2009)
41. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Andoird: Versatile Protection For Smartphones. In: *Proc. 26th Annual Computer Security Applications Conference (ACSAC 2010)*. pp. 347–356 (2010)
42. Sarno, D.: Twitter stores full iPhone contact list for 18 months, after scan (February 2012), <http://articles.latimes.com/2012/feb/14/business/la-fi-tn-twitter-contacts-20120214>
43. Schneider, F.B.: Enforceable Security Policies. *ACM Transactions on Information and System Security* 3(1), 30–50 (2000)
44. von Styp-Rekowsky, P., Gerling, S., Backes, M., Hammer, C.: Callee-site Rewriting of Sealed System Libraries. In: *International Symposium on Engineering Secure Software and Systems (ESSoS'13)*. LNCS, Springer (2013), to appear.
45. Xu, R., Saïdi, H., Anderson, R.: Aurasium – Practical Policy Enforcement for Android Applications. In: *Proc. 21st Usenix Security Symposium* (2012)
46. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In: *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS 2012)* (Feb 2012)